A relaxation of Coq's guard condition

Pierre Boutillier

A relaxation of Coq's guard condition

Pierre Boutillier

Univ Paris Diderot, Sorbonne Paris Cité, PPS, UMR 7126 CNRS, PiR2, INRIA Paris-Rocquencourt, F-75205 Paris, France

Résumé

In a convenient language to handle dependent algebraic data types, this article describes how commutative cuts are used to tackle dependency in pattern matching. It defines a new termination criteria by structural guard condition that allows commutative cuts. Everything exposed scales directly to the Coq proof assistant and describes its implementation. The proof of strong normalisation does not and is still work to do.

Introduction

Proof assistants come with many technical subtleties that any user have to learn before being able to convince a computer of anything non trivial. One example is type checking in Coq where termination checking plays an important role. Termination checking is made up of the application of rules that ensure that one argument of a fixpoint decreases by some measure. These rules are called *guard conditions*.

At such a point, knowing exactly what algorithm is behind the test is the very minimum expected to be able to polish the code in order to make it fit the requirements. Of course, a description is not enough. The task sometime ends to be very acrobatics while an other technical issue asks for proof in a very precise way that the termination checker does not like. The criteria designed should consequently be compatible with techniques from other part of the system.

The Coq proof assistant is able to check guard on recursive function that rely on over recursive function and allow you to get rid of the empty case while you deal with an non empty list but there is no way to do both at the same time. Here is an example that did not work before our relaxation[1]:

```
fix map2 f n (v1 v2: Vect n) ⟶ match (v1,v2) with
|(Vnil,Vnil) ⇒ Vnil
|(Vcons h1 _ t1, Vcons h2 _ t2) ⇒ Vcons (f h1 h2) _ (map2 _ t1 t2) end
```

Rigorously, the situation in Coq is more complicated than the one described here because constructors can take functions as arguments and can be polymorphic. Nevertheless, up to specific conditions on these extra possibilities[2], termination issues are basically the same in the two systems.

Difficulties gathering together the part of an expressive logic are even bigger when you try to prove its consistency. Actually, proofs made by model exhibition deal with everything all-at-once. In Coq, the section of code to trust is well delimited but optimisation that go further than the proved part of the system are not formally described.

This article will neither find a way to make modular proof of consistency nor offer a proof for the whole system but it will describe the exact implementation of a termination checker smart enough to handle programs generated by other part of Coq. The idea is to reduce the term a bit more than

[1] Syntax is explained in detail section 1

[2] Strict positivity of inductive definitions and forbid of polymorphic recursive arguments

what the type preserving reduction would do but it does not depend on other judgements than itself to do so.

Reaching this goal will first require a presentation of the language tackled and its semantic, a survey of methods used elsewhere in the code that impacts termination checking and a exhibition of the termination checking technique used as well as its limitation in its previous introduction.

Afterwards, choice has been made to expose only the final algorithm and not to describe several systems. Anyway, the description is incremental. Consequently, the first part taken alone as well as the two first parts taken alone offers consistent systems by adding structural rules for unwritten constructions.

1. Language

In typed functional programming languages (e.g. Standard ML [19], Haskell [14], Ocaml [15]), it is common to define algebraic data structures such as the unary representation of natural numbers or the unit type.

```
Type Unit := | () : Unit
```

```
type Nat :=
| 0 : Nat
| S : Nat ⟶ Nat
```

```
match t with
| 0 ⇒ S 0
| S n ⇒ n end
```

```
fix double x ⟶ match x with
| 0 ⇒ 0
| S n ⇒ S (S (double n))
 end
```

Figure 1: Examples of the language syntax

Nat is described by the set of constructors that allow to build elements of the type. For instance, 3 is represented as **S** (**S** (**S** **0**)). Elements of these kinds of types are decomposed by pattern matching and fixpoints illustrated in Figure 1.

Available types in these languages sometime limit the user. To be more expressive, different extensions using dependent types have been studied either in the context of type theory (MLTT [17], CIC [8], UTT [16], ...) or in the context of programming languages (GADT [13], Cayenne [3], Dependent ML [24]). Recently, some languages even merge both point of views such as Agda [21] or Epigram [18]. Here, we consider algebraic data-structures with indices [10]. A typical representative of it is the type of list of Boolean values carrying its length:

```
type Vect : Nat ⟶ ⋆ :=
| Vnil : Vect 0
| Vcons : Bool ⟶ forall (n : Nat), Vect n ⟶ Vect (S n)
```

As an example of expressiveness added by the use of indices, we can enforce a list to be non empty and consequently write a function that returns the tail of a list without having to consider the case of an empty list.

```
fun n (v : Vect (S n)) ⟶ match v with | Vcons h n t ⇒ t end
```

Two formal approaches are known to allow the type checking of such expressions.

The lighter one, derived from [7], is used in dependently typed programming language [22, 21]. Type checking of pattern matching is viewed as a coverage analysis. (This relies on unification.)

The second approach is used in the proof assistant Coq [4], whose logic is the Calculus of Inductive Constructions (CIC). It does not depend on unification. The underlying idea is to see pattern-matching as a dependent elimination scheme canonically attached to inductive types. In particular, the predicate on which the scheme is applied is given explicitly to the type-checker as a type annotation. From the

computational point of view, the annotation says how the type gets specialised in the branches of a pattern matching [9].

It leads to see `match n as k in Nat return T with |0 ⇒ a |S m ⇒ b end` as a particular syntax for $\text{case}_{\texttt{fun } k \longrightarrow T}$ a (fun m ⟶ b) n where case_P is a generic combiner of type P 0 ⟶ forall (m: Nat), P (S m) ⟶ forall (k: Nat), P k for objects in Nat.

Nevertheless, Goguen, McBride and McKinna [12] show the two systems have the same theoretical strength except that the first admits Streicher axiom K (or Unicity of Identity Proof) as a reduction rule. In any other situation, any dependent pattern-matching typable in the first system can be simulated in the second system by inserting equality constraints in type annotations.

It is in fact not mandatory to rely on equality constraints. In many cases, a symbolic evaluation can be obtained by inserting appropriate pattern matching in the type annotation itself [6, 5]

```
match v as v' in Vect n' return (match n' with
                                 |0 ⇒ Unit
                                 |S n0 ⇒ Vect n0
                                 end) with
|Vnil ⇒ ()
|Vcons h m t ⇒ t
end
```

Taking advantages of how the coverage analysis is constrained by patterns in the first approach, we introduce a more compact notation written "as x in I $\overrightarrow{\text{p}}$ return T" where

a list of linear patterns ($\overrightarrow{\text{p}}$) over the type of the matched term,

a name (x) to abstract the matched term,

a type (T) abstracted over the first items.

Outside of the match, x is the actual term (v in the example) and $\overrightarrow{\text{p}}$ are matched against the arguments of the type of v resulting in a substitution (here n' is instantiated by m). Inside, constructors whose type does not follow the pattern (below Vnil) are impossible branches. For handling valid constructors (Vcons there), pattern variables are filled with the relevant term. As a result, the complete definition of tail is:

```
match v as v' in Vect (S n') return Vect n' with
|Vnil ⇏
|Vcons h m t ⇒ t
end
```

This `match` represents the application of a customised scheme of type

```
(forall h m t, P m (Vcons h m t)) ⟶ forall n (v: Vect (S n)), P n v
```

Throughout this paper, we consider a tiny language which emphasises the specific subtleties of dependent pattern matching in Coq. It deals with term dependency but neither type dependency nor polymorphism and is similarly expressive as $\lambda\Pi$ or *LF*.

The syntax of this language is split into five categories:

Linear patterns (p) first are required for pattern matching return clause. Their linearity is not syntactically enforced but is required to be able to encode our return clause by a `match` when pattern matching is allowed in types.

Algebraic kinds (K) are either the sort $\star$ or the dependent product of an algebraic type and a K.

Types (T) are algebraic types and dependent products of an algebraic type and a type.

Terms (t) are those Church-style lambda calculus extended with recursive definitions, constructors and pattern matching.

Contexts (Δ) are made of algebraic types definitions and abbreviations for terms.

$$
\begin{array}{lcl}
\mathrm{p} & ::= & x \mid \mathbf{C}\ \overrightarrow{\mathrm{p}} \\
K & ::= & \star \mid \texttt{forall}\ (x\texttt{:}\ \mathbf{I}\ \overrightarrow{t}),\ K \\
T & ::= & \mathbf{I}\ \overrightarrow{t} \mid \texttt{forall}\ (x\texttt{:}\ \mathbf{I}\ \overrightarrow{t}),\ T \\
t & ::= & a \mid x \mid t\ t \mid \texttt{fun}\ x\texttt{:}\ \mathbf{I}\ \overrightarrow{t} \longrightarrow t \mid \texttt{fix}_T\ f\ \overrightarrow{d}\ x \longrightarrow t \mid \mathbf{C} \\
 & & \mid \texttt{match}\ t\ \texttt{as}\ x\ \texttt{in}\ \mathbf{I}\ \overrightarrow{\mathrm{p}}\ \texttt{return}\ T\ \texttt{with}\ \ldots\ \mid \mathbf{C}\ \overrightarrow{y}\ \Rightarrow\ t\ \ldots\ \texttt{end} \\
\Delta & ::= & \varepsilon \mid \Delta\texttt{;}\ \texttt{let}\ a\texttt{=}t\texttt{:}\ T \mid \Delta\texttt{;}\ \texttt{type}\ I\ \texttt{:}\ K\ \texttt{:=}\ \ldots\ \mid \mathbf{C}\texttt{:}\ T\ \ldots
\end{array}
$$

For readability we do not write the return clause, the type of variables or the kinds in type definitions when they are obvious in context. Similarly, we use names for variables but the issues with α-conversions are left implicit. Because a constructor cannot have functions as arguments, our algebraic types enter the category of strictly positive families [20] and can be seen as inductive types.

If $\texttt{let a = t: T}$ occurs in Δ, $\Delta(\mathrm{a})$ denotes t and $\Delta[\mathrm{a}]$ denotes T.

As in standard type theories, type checking is up to evaluation which is defined by the following standard rewriting rules applied anywhere in the term and not only at toplevel:

δ reduction $\quad \mathrm{a} \mapsto \Delta(a)$

β reduction $\quad (\texttt{fun x: T} \longrightarrow \mathrm{u})\ \mathrm{v} \mapsto \mathrm{u}[\mathrm{v}/\mathrm{x}]$

ι reduction $\quad \texttt{match}\ \mathbf{C}_i\ \overrightarrow{u}\ \texttt{as x in}\ \mathbf{I}\ \overrightarrow{\mathrm{p}}\ \texttt{return T with}\ \ldots\ \mid \mathbf{C}_j\ \overrightarrow{x} \Rightarrow v_j\ \ldots\ \texttt{end} \mapsto v_i[\overrightarrow{u/x}]$

μ reduction $\quad (\texttt{fix}_T\ \mathrm{f}\ \overrightarrow{d}\ \mathrm{x} \longrightarrow \mathrm{u})\ \overrightarrow{t}\ (\mathbf{C}\ \overrightarrow{v}) \mapsto \mathrm{u}[\mathbf{C}\ \overrightarrow{v}/\mathrm{x}][\overrightarrow{t/d}][\texttt{fix}_T\ \mathrm{f}\ \overrightarrow{d}\ \mathrm{x} \longrightarrow \mathrm{u}/\mathrm{f}]$

Further than the implicit we make here for readability reason, dependent pattern matching with explicit type annotation syntax is heavy and so is especially the Coq internal syntax. For this reason, the user can employ a lighter language, and, pattern matching thus given is compiled in the inner syntax. This compiler relies on techniques that we will explain in the next section, before exhibiting a solution to handle them.

2. Dependent pattern matching

In dependent settings, two facts complicate pattern matching when compared to the simply-typed case: the types of other objects in the context depend of matched term types and the type of a matched term is not necessarily the one built by constructors of the corresponding inductive type. Here is a survey of how pattern matching is handled in a system based on induction scheme point of view.

When matching the type of the constructor of a branch against the pattern over type given in the `in` clause, three cases can happen:

- The type matches the given pattern and is able to provide a correct answer according to pattern variable instantiation. E.g. **Vcons** in Tail.

- The type is explicitly out of that pattern because two different constructors clash. E.g. **Vnil** in example. In this case, the branch does not answer anything because it cannot correspond to the matched term.
- There is a third case where the given pattern is more precise than the type of the constructor. This would happen, for instance, if we add an extra constructor **Vbig** to **Vect** of type `forall n, Vect (double n)` because Double n can be either **0** or **S ?** depending on n value. This situation is forbidden by our typing rules and should be bypassed by weakening the pattern. With that extra constructor, Nothing better can be done than

```
fun v: Vect S k ⟶ match v
as x in Vect m return P m x with
|Vnil ⇒ ...
|Vcons n0 v0 ⇒ ...
|Vbig n0 ⇒ ... end
```

Discriminating between these cases is done by pattern matching (when allowed in types) using the pattern provided in pattern matching return type clause.

Maintaining the link between types of a matched term and the rest of the context is done by β-expansion around the match. Interleaving β and ι cuts are called *Commutative cuts* and plays a crucial role in our development.

In a simply-typed setting, $(\texttt{match u with} \ldots \mid \mathbf{C}\ \overrightarrow{x} \Rightarrow \texttt{fun y} \longrightarrow \text{f y} \ldots \texttt{end})\ \mathbf{t}$ can be rewritten in $\texttt{match u with} \ldots \mid \mathbf{C}\ \overrightarrow{x} \Rightarrow \text{f t} \ldots \texttt{end}$ and vise-versa. In a dependent setting t and y may not have exactly the same type. So, it must not be reduced.

For example, take two vectors a and b of the same length. If you match over a, in branches, its type will be the one of the constructor. Your only possibility to keep the link between the two types is to put b somewhere in the return clause like

```
(match a as v in Vect n return (Vect n ⟶ [...]) with
|Vnil ⇒ fun b': Vect 0 ⟶ [...]
|Vcons h m t ⇒ fun b': Vect (S m) ⟶ [...] end) b
```

Here is a major reason why commutative cuts are interesting to tackle in the type checker. Coq pattern matching compilation process is based on this transformation but the generated term is refused afterwards by the type checker as soon as recursion is involved.

To be complete, a third smart technique [23] works wonderfully. Generalising equality between types arguments of the matched term and the one of constructors allows to lose no information.

Equality is the special algebraic type that says that everything is equal to itself:

```
type Eq: Nat ⟶ Nat ⟶ ⋆
|eq_refl: forall (n: Nat), Eq n n
```

If u has type **I** t, do

```
match u as z in I x return x = t ⟶ [...] with
... |C y⃗ ⇒ fun eq ⟶ [...] ...
 end (eq_refl t)
```

and eq stores everything.

On the one hand, such terms need dependent equality and consequently K axiom in the general case. On the other hand, proofs over equality block computation in open terms evaluation context, that is used in proofs.

To figure out what we mean on examples, take plus over **Nat** defined by induction on the first argument, then n + **S** m = **S** n + m has to be proved by induction. As a result, when a pattern matching over a **Vect** type-checks in the **Vcons** case thanks to this proof, the ι then β-reduction of (`match` **Vcons** h m t `with` ... `end` (**eq_refl** (S m)) works but you are left with a term (proof_of_plus_Smn_nSm(t)) that does not reduce.

Hopefully, for linear patterns over constructors, which is a common circumstance, everything can be done by pattern matching in the return clause and no explicit equality is required.

3. Structural guard condition

Going back to reduction rules page 4, asking for arguments that begin with a constructor in the rule about fixpoint unfolding is essential. Otherwise, reducing open terms may lead to infinite loops. For instance, we would have the following chain of reduction:

double x $\mapsto$ `match` x `with` | **0** $\Rightarrow$ **0** | **S** n $\Rightarrow$ **S** (**S** (double n)) `end` $\mapsto$

`match` x `with` | **0** $\Rightarrow$ **0** | **S** n $\Rightarrow$ **S** (**S** (`match` n `with` | **0** $\Rightarrow$ **0** | **S** n' $\Rightarrow$ **S** (**S** (double n')) `end`)) `end` $\mapsto$

Yet we need termination for decidability of comparison.

Sized types (see e.g. [2, 1]) and structural guard condition (see e.g. [11]) are two kinds of criteria ensuring termination. We focus on the last one which means that *the number of constructors of a fixpoint argument must strictly decrease at each recursive call.* For instance, double is guarded because the argument of the recursive call has one less constructor than its parent.

Taken alone, the criterion is too limited. To be convinced, consider figure 2. div2 is correct but we will not realise it if app_pred is not unfolded and the obtained expression is not reduced to the last expression which is structurally guarded.

```
let app_pred f x =
match x with
| 0 ⇒ 0
| S n ⇒ f n end
```

```
fix div2 x ⟶ match x with
| 0 ⇒ 0
| S n ⇒  app_pred div2 n end
```

```
fix div2 x ⟶ match x with
| 0 ⇒ 0
| S n ⇒ match n with
| 0 ⇒ 0
| S n' ⇒ div2 n' end end
```

Figure 2: A definition structurally guarded only unfolded

In other words, some β-ι-δ-reduction steps may be needed to reveal that recursive calls are guarded.

In Coq, since the very beginning of CIC implementation, a structural guard condition is used over terms in head normal form to ensure fixpoint termination. Theoretically, a structural guard condition is limited compared to sized types. However, it is challenging to formulate a more expressive approach that can scale up to large examples. Nevertheless, it has three main practical drawbacks that we are going to address:

1. It allows not strictly (but weakly) terminating definitions such as `fix` f x ⟶ (`fun` _ ⟶ **S** **0**) (f **0**), that loop if you eagerly rewrite the arguments but are correct after performing β-reduction steps that erase some subterms.

2. Commutative cuts (defined page 5) are not handled.

   As seen section 2, in a dependent setting, such a reduction rule does not preserve the type and

```
match v in Vect (S k) return Vect k ⟶ Vect k with
|Vnil ⇒ 0
|Vcons h n t ⇒ fun w: Vect n ⟶ w
 end v2
```

must not be reduced to

```
match v in Vect (S k) return Vect k with
|Vnil ⇒ 0 v2
|Vcons h n t ⇒ v2
 end
```

by evaluation.

Though they are safe for termination, only an ad-hoc hack allows some of them in Coq version 8.4. They are not properly supported and this is the motivation of the current work.

3. Commutative cuts can also involve the name of the fixpoint itself, as in `fix g n ⟶ (match n with |0 ⇒ ()|S x ⇒ fun f ⟶ f x end) g` that should be considered guarded.

   Though this example is artificial, it does correspond to a common Coq failure which is not understood by users when writing tactic-defined fixpoints.

A guard condition for our system that scales up directly to Coq will now be described incrementally. First, we will describe rules that we always find when testing structural guards. Afterwards, we will add judgements to simulate ι and μ. In the end, we will show how the remaining rules end to build our own reduction machine for β, and exhibit a more permissive guard condition that ensures strong normalisation.

4. Basis of structural guard condition

To begin with, this part describes the rules of fundamental cases of a structural guard condition. After evaluation of the fixpoint body, these rules are sufficient. Consequently, it draws a correct picture of what was done in Coq up to version 8.3. Our added value will come in the next sections.

To check a structural guard condition, two kinds of judgements are needed. The first one written $\Gamma \vdash_f t \blacktriangleright a$ gives the *specification of subterm* of a term t using

a specification of subterm a , for now, either

- $<$ which stands for subterm or
- $\leq$ for recursive argument or
- $\bot$ for anything else,

an environment Γ to give the specification of subterm of each free variable of t and

the name f of the checked fixpoint.

Then, the main judgement *guarded definition*, written $\Gamma \vdash_f t \#$ for terms and $\Gamma \vdash_f T \#$ for types, expresses that every recursive call to f is guarded in t or T respectively.

Concretely, ensuring that the recursive call to f is guarded in the definition $\texttt{fix}_T\ f\ \overrightarrow{d}\ x \longrightarrow t$ reduces to finding a derivation whose conclusion is $\overrightarrow{d \blacktriangleright \bot}, x \blacktriangleright \leq \vdash_f t \#$ using the following rules:

- An applied constructor or inductive type is guarded if every of its arguments is guarded.
- A recursive call is guarded iff its last argument has the specification $<$ and all its arguments are guarded.
- A variable is guarded and has the specification it has in Γ.
- A dependent product, as any type, is never a subterm. Hence there is neither a rule IndSpec nor ProdSpec one. Guard check consists in recursive calls.
- More interesting, in the case of match, we define a *strengthen* operation ($\|\_\triangleright\|$) on specifications of subterm by $\|<\| = \| \leq \| = <$ and $\|\bot\| = \bot$. We look for the specification of the matched term and we affect its strength version to each variable of the pattern.

  Of course, all pattern variables are not allowed as recursive arguments but typing handles it for us.
- In addition, to get the specification of subterm of a match expression, we look for the specification of every branch and concatenate them.

  Consequently, we introduce an operation Π over specifications where $\bot$ is a zero, and $<$ a unit.

$$\textsc{Constructor}\ \frac{\ldots\ \Gamma \vdash_f t\ \#\ \ldots}{\Gamma \vdash_f \mathbf{C}\ \ldots\ t\ \ldots\ \#} \qquad \textsc{Ind}\ \frac{\ldots\ \Gamma \vdash_f t\ \#\ \ldots}{\Gamma \vdash_f I\ \ldots\ t\ \ldots\ \#}$$

$$\textsc{Fix}\ \frac{\ldots\ \Gamma \vdash_f u\ \#\ \ldots \quad \Gamma \vdash_f t\ \# \quad \Gamma \vdash_f t \blacktriangleright <}{\Gamma \vdash_f f\ \overrightarrow{u}\ t\ \#} \qquad \textsc{VarSpec}\ \frac{}{\Gamma \vdash_f x \blacktriangleright \Gamma(x)}$$

$$\textsc{Var}\ \frac{x \neq f}{\Gamma \vdash_f x\ \#} \qquad \textsc{Prod}\ \frac{\ldots\ \Gamma \vdash_f t\ \#\ \ldots \quad \Gamma \vdash_f T\ \#}{\Gamma \vdash_f \texttt{forall}\ (x\texttt{:}\mathbf{I}\,\overrightarrow{t})\texttt{,}\ T\ \#}$$

$$\textsc{Match}\ \frac{\Gamma, \overrightarrow{y \in \mathbf{p} \blacktriangleright \bot}, z \blacktriangleright \bot \vdash_f T\ \# \quad \Gamma \vdash_f t \blacktriangleright a \quad \ldots\ \Gamma, x \blacktriangleright \overrightarrow{\|a\|} \vdash_f b_i\ \#\ \ldots}{\Gamma \vdash_f \begin{array}{l}\texttt{match}\ t\ \texttt{as}\ z\ \texttt{in}\ \mathbf{I}\ \overrightarrow{\mathbf{p}}\ \texttt{return}\ T \\ \texttt{with}\ \ldots\ \texttt{|}\,\mathbf{C}_i\ \overrightarrow{x}\ \Rightarrow\ b_i\ \ldots\ \texttt{end}\end{array}\ \#}$$

$$\textsc{MatchSpec}\ \frac{\Gamma \vdash_f t \blacktriangleright a \quad \ldots\ \Gamma, \overrightarrow{x \blacktriangleright \|a\|} \vdash_f u_i \blacktriangleright a_i\ \ldots \quad \prod_i a_i = b}{\Gamma \vdash_f \texttt{match}\ t\ \texttt{with}\ \ldots\ \texttt{|}\,\mathbf{C}_i\ \overrightarrow{x}\ \Rightarrow\ u_i\ \ldots\ \texttt{end} \blacktriangleright b}$$

$$\textsc{FixSpec}\ \frac{\Gamma \vdash_f v \blacktriangleright b \quad \Gamma, g \blacktriangleright \bot, \overrightarrow{d \blacktriangleright \bot}, x \blacktriangleright b \vdash_f u \blacktriangleright a}{\Gamma \vdash_f \left(\texttt{fix}_{T'}\ g\ \overrightarrow{d}\ x \longrightarrow u\right)\ \overrightarrow{t}\ v \blacktriangleright a}$$

Standard rule of a syntactic structural guard condition

- While looking for the recursive call of f in the body of g, we can affect the specification of v to x since a recursive argument length can only decrease. Remark that guard checks are made one after the other and g correctness is not mixed with the one of f.

5. Be more careful with inductive eliminators

In the previous implementation, a fixpoint's body was reduce to its normal form before being checked. rules presented until now were consequently enough. Cuts are now handled manually. A lot of bureaucracy is added but, as we will see in Section 6, simulating reduction by ourselves increases the power of the system. To begin with, we tackle ι and μ reductions.

Of course, an applied constructor is not and shall not be a subterm in order to keep things finite. Relaxing the condition is incompatible with the fact that a fixpoint unfolds itself if its argument starts by a constructor (see page 6).

But an applied constructor is not a term like the others. Imagine that you end with

```
match S n with |0 ⇒ evil |S m ⇒ good end
```

You would be glad to mark **evil** as dead code and only look for the **good** branch.

For this purpose, rules involving inductive constructions become more than strict recursive calls.

Before that, the specification of subterm data-structure becomes recursive and is extended with a notion of constructor specification:

Constructor $\mathbf{C}$ whose arguments gives the specifications $\overrightarrow{a}$ of the arguments of the constructor.

In the same time, Π has to be extended as expected. For anything but the exact same constructor $\mathbf{C}$, $\mathbf{C}\ \overrightarrow{\mathrm{a}} \cdot s = \bot$ and $\mathbf{C}\ \overrightarrow{\mathrm{a}} \cdot \mathbf{C}\ \overrightarrow{\mathrm{b}} = \mathbf{C}\ \overrightarrow{a \cdot b}$

Then, judgements have the material to reduce ι-μ cuts. Actually, the applied constructor C_i is and is the only kind of term to be a term of specification C_i. As a result, substituting a pattern-matching over a term with specification $C_i\ \overrightarrow{a}$ by its C_i branch or a fixpoint whose argument's specification has the same structure by its body are exactly the simulation of the reduction rules in the judgement.

$$\textsc{ConstructorSpec}\ \frac{\ldots\ \Gamma \vdash_f t \blacktriangleright a\ \ldots}{\Gamma \vdash_f \mathbf{C}\ \overrightarrow{t} \blacktriangleright \mathbf{C}\ \overrightarrow{a}} \qquad \textsc{Iota}\ \frac{\Gamma \vdash_f u \blacktriangleright \mathbf{C}_i\ \ldots\ a\ \ldots \qquad \Gamma, \ldots\ x \blacktriangleright a\ \ldots \vdash_f t_{C_i}\ \# \qquad \ldots\ \Gamma, \ldots\ x \blacktriangleright <\ \ldots \vdash_f t_{C_j}\ \#\ \ldots}{\Gamma\ \vdash_f\ \texttt{match}\ u\ \texttt{with}\ \ldots\ |\mathbf{C}\overrightarrow{x}\ \Rightarrow\ t\ \ldots\ \texttt{end}\ \#}$$

$$\textsc{Fix'}\ \frac{\Gamma \vdash_f v \blacktriangleright \mathbf{C}\ \overrightarrow{s} \qquad \ldots\ \Gamma \vdash_f u \blacktriangleright b\ \ldots \qquad \Gamma, \overrightarrow{d \blacktriangleright b}, x \blacktriangleright \mathbf{C}\ \overrightarrow{s} \vdash_f t\ \#}{\Gamma\ \vdash_f\ f\ \overrightarrow{u}\ v\ \#}\ \text{for}\ \mathtt{fix}_T\ \mathrm{f}\ \overrightarrow{d}\ \mathrm{x} \longrightarrow \mathrm{t}$$

Figure 3: ι-μ simulation without explicit reductions

6. The machinery

We now focus on the way to check guard up to β-equivalence. We achieve β-reduction through the use of ad-hoc rules in order allow more definition.

The judgements are enriched as it is commonly done in an abstract machine of

a substitution σ where "values" of variable will be stored *if necessary* for future check. We will explain what we mean at the end of the section.

a stack s to store arguments between the moment where we tackle the application and the related lambda. Actually, there can be a pattern matching between application and the abstraction. This is the exact point where commutative cuts are applied.

By values and arguments we actually mean closures, to avoid problems with variable capture.

Syntax of full judgements are

$$\boxed{(\Gamma,\ \sigma) \vdash_f s \mid t \blacktriangleright a} \qquad \boxed{(\Gamma,\ \sigma) \vdash_f s \mid t\ \#}$$

Rules given until here are the same except that they carry s and σ without any other modification.

Remaining behaviours to explain are:

- Application feeds the stack. This way, check of arguments are delayed to time of use.
- Consequently, a defined variable must call the check of its value.
- A lambda transfers from a non empty stack to the substitution.
- If the stack is empty, a variable is declared instead of being defined.

If we stop here, the non terminating example in call by value (item 1 p.6) is allowed again! That is why all computation over right part of application cannot be completely delayed. The rule **sanity** accomplishes this. It hides the control of v in a new variant of check fixpoint drawn $\boxed{(\Gamma,\ \sigma) \vdash_f s \mid v \approx}$ where

$$\textsc{App}\ \frac{(\Gamma,\ \sigma) \vdash_f (\Gamma, \sigma, v) :: s \mid u\ \#}{(\Gamma,\ \sigma) \vdash_f s \mid u\ v\ \#}\ \textbf{sanity}$$

$$\textsc{Def}\ \frac{\sigma(y) = (\Gamma', \sigma', t) \qquad (\Gamma',\ \sigma') \vdash_f s \mid t\ \#}{(\Gamma,\ \sigma) \vdash_f s \mid y\ \#}$$

$$\textsc{Pop}\ \frac{(\Gamma,\ \sigma) \vdash_f \emptyset \mid c \blacktriangleright a \qquad (\Gamma, x \blacktriangleright a,\ \sigma, x \mapsto c) \vdash_f s \mid u\ \#}{(\Gamma,\ \sigma) \vdash_f c :: s \mid \texttt{fun}\ x \longrightarrow u\ \#}$$

$$\textsc{Lam}\ \frac{(\Gamma, x \blacktriangleright \bot,\ \sigma, x) \vdash_f \emptyset \mid t\ \#}{(\Gamma,\ \sigma) \vdash_f \emptyset \mid \texttt{fun}\ x : T \longrightarrow t\ \#}$$

Abstract machine embedded in rule set

- a new specification of subterm called Neutral and written $\diamond$ is added.
- Lam rule is modified to add to Γ a $\diamond$ instead of a $\bot$
- Fix asks for recursive call to be $<$ or $\diamond$.

Informally, it means to be "safely reducible in an empty context".

Moreover, the weak guard condition where every recursive call specification is strict subterm "$<$" implies the strong guard. This way, while checking **sanity** in App, we could implement the optimisation were a strictly guarded argument is declared and not defined. It is consequently checked only once even if it is used more than once.

$$\textsc{StrongApp}\ \frac{(\Gamma,\ \sigma) \vdash_f \emptyset \mid v\ \# \qquad (\Gamma,\ \sigma) \vdash_f \_ :: s \mid u\ \#}{(\Gamma,\ \sigma) \vdash_f s \mid u\ v\ \#}$$

The end of our journey is now reached. By making the stack explicit, it can be given to the branches of a match and simulate commutative cuts.

$$\textsc{Match}\ \frac{\begin{array}{c}\ldots\ (\Gamma, \overrightarrow{x \blacktriangleright \|a\|},\ \sigma, \overrightarrow{x}) \vdash_f \mathbf{s} \mid b_i\ \#\ \ldots \\ (\Gamma,\ \sigma) \vdash_f \emptyset \mid t \blacktriangleright a \qquad (\Gamma, \overrightarrow{y \in \mathbf{p} \blacktriangleright \bot}, z \blacktriangleright \bot,\ \sigma, \overrightarrow{y \in \mathbf{p}}, z) \vdash_f \emptyset \mid T\ \#\end{array}}{(\Gamma,\ \sigma) \vdash_f s \mid \texttt{match}\ t\ \texttt{as}\ z\ \texttt{in}\ \mathbf{I}\ \overrightarrow{\mathbf{p}}\ \texttt{return}\ T\ \texttt{with}\ \ldots\ \texttt{|}\mathbf{C}_i\ \overrightarrow{x}\ \Rightarrow\ b_i\ \ldots\ \texttt{end}\ \#}$$

Figure 4: Commutative cuts in syntactic guard condition

7. Typing

At last, termination checking is part of the type checking process. In order to get the general picture, we present its rules under the hypothesis that $\_ =_{\beta\delta\iota\mu} \_$ stands for being equals modulo evaluation and that (parallel) substitution deals correctly with α-conversion:

- For terms:

$$\textsc{Def}\ \frac{}{(\Delta;\Gamma)\vdash a : \Delta[a]} \qquad \textsc{Var}\ \frac{}{(\Delta;\Gamma)\vdash x : \Gamma(x)} \qquad \textsc{App}\ \frac{(\Delta;\Gamma)\vdash u : \texttt{forall}\ (x\texttt{:}\,\mathbf{I}\,\overrightarrow{t})\texttt{,}\ T \qquad (\Delta;\Gamma)\vdash v : \mathbf{I}\,\overrightarrow{t}}{(\Delta;\Gamma)\vdash u\ v : T[v/x]}$$

$$\textsc{Lam}\ \frac{(\Delta;\Gamma,x\texttt{:}\,\mathbf{I}\,\overrightarrow{t})\vdash u : T \qquad (\Delta;\Gamma)\vdash \mathbf{I}\,\overrightarrow{t} \in \star}{(\Delta;\Gamma)\vdash (\texttt{fun}\ x\texttt{:}\,\mathbf{I}\,\overrightarrow{t}\longrightarrow u) : \texttt{forall}\ (x\texttt{:}\,\mathbf{I}\,\overrightarrow{t})\texttt{,}\ T} \qquad \textsc{Constr}\ \frac{}{(\Delta;\Gamma)\vdash \mathbf{C} : \Delta(\mathbf{C})}$$

$$\textsc{Match}\ \frac{\begin{array}{c}\Delta(I) = \texttt{forall}\ \overrightarrow{w\texttt{:}\,S}\texttt{,}\ \star \qquad \Delta\vdash \overrightarrow{S} \ni \overrightarrow{\mathbf{p}} \triangleright \overrightarrow{L} \qquad (\Delta;\Gamma)\vdash \texttt{forall}\ \overrightarrow{L}\texttt{,}\ T\in\star \\ (\Delta;\Gamma)\vdash t : \mathbf{I}\overrightarrow{a} \qquad \ldots\ \Delta(\mathbf{C}) = \texttt{forall}\ \overrightarrow{x\texttt{:}\,U}\texttt{,}\ \mathbf{I}\,\overrightarrow{b} \qquad (\Delta;\Gamma,\overrightarrow{x\texttt{:}\,U})\vdash u : T[\overrightarrow{b}/\overrightarrow{\mathbf{p}}][\mathbf{C}\,\overrightarrow{x}/z]\ \ldots\end{array}}{(\Delta;\Gamma)\vdash \texttt{match}\ t\ \texttt{as}\ z\ \texttt{in}\ \mathbf{I}\overrightarrow{\mathbf{p}}\ \texttt{return}\ T\ \texttt{with}\ \ldots\ \texttt{|}\,\mathbf{C}\ \overrightarrow{x}\ \Rightarrow\ u\ \ldots\ \texttt{end} : T[\overrightarrow{a}/\overrightarrow{\mathbf{p}}][t/z]}$$

$$\textsc{Fix}\ \frac{(\Delta;\Gamma,f\texttt{:}\,T)\vdash \texttt{fun}\ \ldots\ d\ \ldots\ x\longrightarrow t : T \qquad (\overrightarrow{d\blacktriangleright\bot},x\blacktriangleright\leq,\ \Delta)\vdash_f \emptyset\mid t\ \#}{(\Delta;\Gamma)\vdash (\texttt{fix}_T\ f\ \overrightarrow{d}\,x\longrightarrow t) : T}$$

Two notations are not standard. First, in the `fix` case, types of d and x must be extracted from T but writing it explicitly would only obfuscate the rule.

More important is the pattern substitution in the `match` rule. If the pattern **p** is a variable, it is the common substitution. If the pattern is a constructor and the term begins by the same constructor, it is recursive calls. If the constructor of b is different, u if exists has to be well typed in an arbitrary type. This is the impossible branch case. If **p** starts with a constructor and b is not an applied constructor then type checking fails (see p.5).

- For types and kinds:

$$\textsc{Ind}\ \frac{\Delta(\mathbf{I}) = \texttt{forall}\ \overrightarrow{x\texttt{:}\,T}\texttt{,}\ \star \qquad \ldots\ (\Delta;\Gamma)\vdash t_i : T_i[t_j/x_j]_{1\ldots i}\ \ldots}{(\Delta;\Gamma)\vdash \mathbf{I}\,\overrightarrow{t}\ \in\star} \qquad \textsc{Conv}\ \frac{S =_{\beta\delta\iota\mu} T \qquad (\Delta;\Gamma)\vdash S\in\star}{(\Delta;\Gamma)\vdash T\in\star}$$

$$\textsc{Prod}\ \frac{(\Delta;\Gamma)\vdash \mathbf{I}\,\overrightarrow{t}\in\star \qquad (\Delta;\Gamma,x\texttt{:}\,\mathbf{I}\,\overrightarrow{t})\vdash T\in\star}{(\Delta;\Gamma)\vdash \texttt{forall}\ (x\texttt{:}\,\mathbf{I}\,\overrightarrow{t})\texttt{,}\ \mathbf{T}\in\star} \qquad \textsc{Star}\ \frac{}{(\Delta;\Gamma)\vdash\star} \qquad \textsc{ForAll}\ \frac{(\Delta;\Gamma)\vdash \mathbf{I}\,\overrightarrow{t}\in\star \qquad (\Delta;\Gamma,x\texttt{:}\,\mathbf{I}\,\overrightarrow{t})\vdash K}{(\Delta;\Gamma)\vdash \texttt{forall}\ (x\texttt{:}\,\mathbf{I}\,\overrightarrow{t})\texttt{,}\ K}$$

- For contexts:

$$\textsc{Empty}\ \frac{}{\emptyset\ \dashv} \qquad \textsc{Cons}\ \frac{\Delta\ \dashv \qquad (\Delta;\emptyset)\vdash t : T}{\Delta;\texttt{let}\ a\ \texttt{:=}\ t\texttt{:}\quad T\ \dashv}$$

$$\textsc{Type}\ \frac{\Delta\ \dashv \qquad (\Delta;\emptyset)\vdash K \qquad \ldots\ T = \texttt{forall}\ \_\texttt{,}\ \mathbf{I}\_(\Delta;\texttt{type}\ I\texttt{:}\ K;\emptyset)\vdash T\in\star\ \ldots}{\Delta;\texttt{type}\ I\texttt{:}\quad K\texttt{:=}\ \ldots\ \mathbf{C}\texttt{:}\quad T\ \ldots\ \dashv}$$

- And for patterns

$$\textsc{Nil}\ \frac{}{\Delta \vdash \emptyset \ni \emptyset \triangleright \emptyset} \qquad \textsc{Var}\ \frac{\Delta \vdash \overrightarrow{T} \ni \overrightarrow{\mathbf{p}} \triangleright \overrightarrow{L}}{\Delta \vdash U :: \overrightarrow{T} \ni x :: \overrightarrow{\mathbf{p}} \triangleright (x : U) :: \overrightarrow{L}}$$

$$\textsc{Rec}\ \frac{\Delta(\mathbf{C}) = \texttt{forall}\ \overrightarrow{x : T},\ \mathbf{I}\,\overrightarrow{t} \qquad \Delta \vdash \overrightarrow{T} :: \overrightarrow{T'} \ni \overrightarrow{\mathbf{p}} :: \overrightarrow{\mathbf{p}'} \triangleright \overrightarrow{L}}{\Delta \vdash \mathbf{I}\,\overrightarrow{t} :: \overrightarrow{T'} \ni \mathbf{C}\overrightarrow{\mathbf{p}} :: \overrightarrow{\mathbf{p}'} \triangleright \overrightarrow{L}}$$

Conclusion

Ensuring termination raises a lot of technical issues. We have shown here how to take better advantage of one elementary criterion: the syntactic structural guard condition.

The main advantage of the presented work is to remove situations where the automatic part of proof term generation ends to functions refused by the Coq termination checker.

We have exhibited that enhancing the regular syntactic guard condition judgement with one ad-hoc reduction machine is the key to get a more powerful and stand alone rules set.

Formal correctness must be the next step. The two known techniques to do so are reducibility candidate and relative soundness by encoding. Nevertheless, these proofs are system specific. A result for our playground is just the beginning of the Coq case, and the CIC is very heavy to manipulate at once. An ideal proof that scales up would consequently say: suppose that a well typed and guarded term has no infinite β reduction chain then it has no β-ι-μ reduction chain.

Judgements have been defined in order to ensure that :

- ι naturally "consumes" a constructor and we do not deal with higher order in constructor's argument so no β-ι infinite chain is possible.
- A guarded term has no infinite ι-μ-reduction chain
- Guard and typing are stable by β-ι-μ reduction. Consequently, a well typed guarded term never evaluates to a term that has an infinite β-ι reduction chain or a ι-μ reduction chain.

A statement like "a β-ι-μ infinite chain implies an β-ι infinite chain or a ι-μ one" would complete syntactically our proof but is absolutely not straight forward.

More deeply, our algorithmic complexity is worse than typed base criteria and information is still lost when a fixpoint has more than one argument that decrease. Only one is taken into account and can be used in mutual fixpoints definition.

Bibliographie

[1] Andreas Abel. Termination checking with types. *RAIRO – Theoretical Informatics and Applications*, 38(4):277–319, 2004. Special Issue: Fixed Points in Computer Science (FICS'03).

[2] Andreas Abel and Thorsten Altenkirch. A predicative analysis of structural recursion. *Journal of Functional Programming*, 12(1):1–41, January 2002.

[3] Lennart Augustsson. Cayenne a language with dependent types. In *Proceedings of the third ACM SIGPLAN international conference on Functional programming*, ICFP '98, pages 239–250, New York, NY, USA, 1998. ACM.

[4] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer Verlag, 2004.

[5] Pierre Boutillier. *Cloisonnement des contenus calculatoire et logique du filtrage dépendant en théorie des types et application aux coupures commutatives*. Master thesis, Université Paris Diderot, 2010.

[6] Adam Chlipala. An introduction to programming and proving with dependent types in Coq. *Journal of Formalized Reasoning*, 3(2):1–93, 2010.

[7] Th. Coquand. Pattern Matching with Dependent Types. In B. Nordström, K. Petersson, and G. Plotkin, editors, *Proceedings of the 1992 Workshop on Types for Proofs and Programs*. Available by ftp at site ftp.inria.fr, 1992.

[8] Th. Coquand and C. Paulin-Mohring. Inductively defined types. In P. Martin-Löf and G. Mints, editors, *Proceedings of Colog'88*, volume 417 of *Lecture Notes in Computer Science*. Springer-Verlag, 1990.

[9] Thierry Coquand and Peter Dybjer. Inductive definitions and type theory: an introduction (preliminary version). In *FSTTCS*, pages 60–76, 1994.

[10] Peter Dybjer. Inductive families. *Formal Aspects of Computing*, 6:440–465, 1997.

[11] Eduardo Giménez. Structural recursive definitions in type theory. In *ICALP*, pages 397–408, 1998.

[12] Healfdene Goguen, Conor McBride, and James McKinna. Eliminating dependent pattern matching. In *Essays Dedicated to Joseph A. Goguen*, pages 521–540, 2006.

[13] Haskell wiki on generalized algebraic datatypes. See http://www.haskell.org/haskellwiki/GADT.

[14] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, May 2003.

[15] Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The Objective Caml system, documentation and user's manual – release 3.12*. INRIA, August 2010.

[16] Zhaohui Luo. *Computation and reasoning: a type theory for computer science*. Oxford University Press, Inc., New York, NY, USA, 1994.

[17] Per Martin-Löf. An intuitionistic theory of types.

[18] Conor McBride. Epigram, 2004. `http://www.dur.ac.uk/CARG/epigram`.

[19] Robin Milner, Mads Tofte, Robert Harper, and David Macqueen. *The Definition of Standard ML - Revised*. The MIT Press, rev sub edition, May 1997.

[20] Peter Morris, Thorsten Altenkirch, and Neil Ghani. Constructing strictly positive families. In Joachim Gudmundsson and Barry Jay, editors, *Thirteenth Computing: The Australasian Theory Symposium (CATS2007)*, volume 65 of *CRPIT*, pages 111–121, Ballarat, Australia, 2007. ACS.

[21] Ulf Norell. Dependently typed programming in Agda. In *In Lecture Notes from the Summer School in Advanced Functional Programming*, 2008.

[22] Brigitte Pientka and Joshua Dunfield. Beluga: A framework for programming and reasoning with deductive systems (system description). In *IJCAR*, pages 15–21, 2010.

[23] Matthieu Sozeau. *Un environnement pour la programmation avec types dépendants*. PhD thesis, Université Paris 11, Orsay, France, December 2008.

[24] Hongwei Xi. Dependent ML an approach to practical programming with dependent types. *J. Funct. Program.*, 17:215–286, March 2007.